Pràctica 1 - eSportsLS

Ordenació mitjançant recursivitat

Marti Ejarque Galindo (marti.ejarque)
Victor Xirau Guardans (victor.xirau)

December 2019

Contents

1 Introducció

En aquesta pràctica se'ns ha plantejat programar mètodes d'ordenació, en un llenguatge de programació a escollir, de la informació de diferents arxius JSON, relacionada amb el videojoc League of Legends. També se'ns han donat indicacions sobre quins criteris d'ordenació s'han de seguir, relacionats amb les dades proporcionades de cada equip, i els mètodes d'ordenació a implementar, mostrats a continuació.

1.1 Mètodes d'ordenació

S'han programat quatre diferents mètodes d'ordenació, on cada un té característiques diferents però el mateix objectiu, ordenar segons un criteri preestablert un cert llistat d'informació.

- Quick Sort:

Aquest mètode d'ordenació consisteix en anar dividint recursivament el nostre problema, tot escollint un pivot y ordenant-lo segons el criteri preestablert a cada volta de la recursivitat. Un cop ordenat aquest pivot, es torna a escollir un nou pivot per ordenar-lo de nou, i així fins a tenir tot el llistat ordenat segons el criteri.

- Merge Sort:

Aquest mètode d'ordenació consisteix en anar dividint el input en subproblemes iguals al input, i un cop separats en casos trivials (segons un criteri establert), s'ordenen i es tornen a ajuntar ordenats. Per aconseguir aquest procés, per cada subproblema es va trobant la meitat, el inici i el fi. D'aquesta manera es pot anar dividint recursivament fins arribar al moment desitjat per poder ordenar i seguidament anar unint les parts separades ja ordenades.

- Bucket Sort:

Aquest mètode d'ordenació consisteix en col·locar l'input en diferents "cubells", que representaran una particularitat de l'input, és a dir, si tenim números com a input, els "cubells" representaran els dígits, on cada "cubell" serà un dígit, i es posaran els números en el "cubell" que li pertoqui. Un cop fet aquesta selecció, s'ordena cada "cubell", seguint el mateix procés, fins que jo no queden dígits per separar.

- Radix Sort:

Aquest mètode d'ordenació es sol aplicar només a números enters. Consisteix en mirar cada dígit, de menys significatiu a més o viceversa, i anar ordenant cada dígit fins arribar al costat contrari d'on s'ha començat. Aquest mètode també funciona per números de punt flotant, tot i que la metodologia d'operar es lleugerament diferent, degut a la coma d'aquests números.

2 Codificació del codi

En aquest apartat de la memòria s'exposarà com s'ha codificat cada un dels mètodes d'ordenació demanats en la pràctica, així com una justificació de quin llenguatge de programació s'ha utilitzat.

2.1 Llenguatge de Programació

En aquesta pràctica, s'ha decidit utilitzar Java com a llenguatge de programació orientada a objectes per diferents motius exposats a continuació:

- Facilitat en el codi:

Sent alumnes de Segon Curs, hem vist com el pas de C a Java ha estat molt atractiu, ja que compte amb un llenguatge simple i compacte, que ens ha ajudat a l'hora de fer certs aspectes del nostre programa, ja sigui per la senzillesa del llenguatge com les llibreries que et deixa importar, que faciliten el fet de no haver d'implementar certes parts del codi, com la creació de Llistes o Arrays dinàmics.

- Importació de llibreries:

Gràcies a la gran quantitat de llibreries amb les que conta Java, s'ha pogut contar amb moltes eines necessàries per fer les ordenacions apropiadament .Entre les que s'ha utilitzat, podem destacar algunes com la llibreria amb tot el relacionat amb Arrays dinàmics, la llibreria Gson, que ens ha permès llegir de manera molt còmode els diferents fitxers JSON de la pràctica, entre altres.

- Orientat a objectes:

Una de les grans comoditats respecte a C que té Java és la creació de classes. Gràcies a aquesta creació de classes, la programació es fa més intuïtiva. En aquestes classes es creen objectes, que tenen dins diferent informació i funcions que ens permeten manipular i consultar informació amb més facilitat. Gràcies a aquesta orientació a objectes, el fet de llegir un JSON es fa menys tediós, ja que amb una llibreria i les classes adequades, tens tota la informació d'aquest fitxer al teu abast.

- Gestió d'errors:

Tot i que és cert que l'entorn de desenvolupament ajuda en la detecció d'errors, Java conta amb una gestió més potent y específica que no pas C o C++. Aquesta gestió d'errors et dona més detalls sobre quin es l'error que té el codi, mitjançant les "Exceptions". Aquestes seran "llançades" amb una descripció que ens ajudarà més a comprendre quin ha estat l'error comès.

També cal mencionar que hem fet servir un entorn de desenvolupament integrat de Java, InteliJ Idea. Amb aquesta eina hem pogut compilar i anar veient els diferents errors i problemàtiques ocasionades pels diferents mètodes. Aquesta eina, juntament amb l'editor de text Atom, ens han permès treballar anàlogament durant el transcurs de la pràctica, podent editar codi a la vegada.

2.2 Codificació

En aquest apartat de la memòria s'exposarà com s'ha codificat els diferents mètodes d'ordenació, així com els criteris utilitzats a l'hora de crear el codi. Abans d'exposar cada mètode, cal fer una menció a com està distribuïda la pràctica, és a dir, les diferents classes creades i la seva funcionalitat.

La pràctica està dividida en diferents classes, on cada una té un paper important a l'hora de fer funcionar els diferents mètodes d'ordenació.

- Main.java:

Aquesta classe s'ocupa de llegir el fitxer JSON que l'usuari seleccioni. Un cop llegit, se li pregunta el mètode d'ordenació que vulgui utilitzar i seguidament el criteri pel que s'ordenarà el fitxer. Un cop triades aquestes dues opcions, el Main crida aquelles funcions encarregades a dur a terme la tasca demanada, que es troben en una altra classe. El Main també s'encarrega de printar per pantalla els resultats de les opcions demanades i donar a conèixer al usuari el cost temporal de les opcions seleccionades.

- Equip.java:

Aquesta classe s'encarrega de crear tota la estructura necessària per guardar la informació del JSON, és a dir, es creen tots els atributs que formen un

equip. Dins d'aquesta classe també s'ha inclòs tot el relacionat amb la lògica i codificació dels mètodes d'ordenació anteriorment mencionats.

- Jugador.java:

Aquesta classe s'encarrega de crear la estructura necessària perquè el JSON es pugui guardar correctament, és a dir, crea tots els atributs que ha de tenir un jugador. En aquesta classe també hi ha els Getters i Setters, per poder inicialitzar o consultar informació sobre jugadors. També cal dir que s'ha associat el winrate del equip al jugador, d'aquesta manera no s'ha de consultar l'equip d'aquest jugador sempre que es vulgui saber el seu winrate, sinó que directament es consulta des del jugador.

- Leyenda.java:

Aquesta classe simplement crea l'estructura per poder guardar un camp concret de jugador, el seu personatge que més utilitza.

2.2.1 Quick Sort

Per aconseguir aquest mètode d'ordenació, s'ha dividit el problema en dues branques diferents, segons el criteri a escollir per l'usuari. Per tant, s'han fet dos Quick Sorts diferents, un que ordena Equips, i un altre que ordena Jugadors.

- Quick Sort Equip (Per WinRate):

En aquest mètode d'ordenació, primer de tot es crida la funció "partition". Aquesta funció s'encarrega d'establir quin serà el pivot, és a dir, el Winrate que s'utilitzarà com a referència. Un cop establert, es recorre l'array des del índex start fins l'end, comparant un per un si el WinRate que es consulta és més gran que el del pivot. De ser així es fa un swap i la variable start augmenta (cosa que ens dirà que aquella part del array ja ha estat ordenat), per així saber quin serà la porció del array d'equips que encara no està ordenada.
Un cop acabada la criada a "partition" es mira si el que retorna aquesta funció (l'índex start) és més gran que l'start original. De ser així, es torna a fer el procés anteriorment explicat, però des del start inical fins al nou start que ens retorna la funció "partition".

- Quick Sort Jugador (Per Nacionalitat):

Per aquest segon criteri, de igual manera que pel primer, es crida a la funció partition, amb la petita diferencia que per la nacionalitat hem hagut de crear una funció partition a part que hem anomenat partitionNationality. Aquesta funciona de manera similar a la anterior mencionada funció partition de la funció quickSort Equip. La funció partition nationality te com a objectiu comparar el array amb el pivot i endreçar-lo respecte el mateix. Però, mentre que pel winrate estem comprant nombres, per poder endreçar-ho per nacionalitat aquí fem us de la eina compareTo pròpia de la classe String de java. Aquesta ens retorna un nombre negatiu en cas que el string a vagi després que el string b al abecedari. És per això que en cas que això es compleixi fem el swap pertinent per canviar de ordre els dos strings. En cas que els dos strings siguin iguals, per tant que siguin de la mateixa nacionalitat. El enunciat ens demana que comparem en funció del nom. Per fer això tornem a fer us de la eina compareTo però també forcem que els compari tots en "lowerCase" perquè hi ha noms iguals però que poden tenir majúscules a diferents llocs i, aparentment, la funció compareTo els endreça en funció del ascii i les majúscules van abans al ascii que les minúscules. Un cop comparats els noms apliquem la mateixa lògica que per les nacionalitats, si el resultant de compareTo es negatiu es realitza el switch.
Un cop acabada la crida partition es retorna, igual que en l'apartat anterior, la posició del inici del array que s'acaba de endreçar.

- Quick Sort Jugador (Per Combinació de Prioritats):

Per aquest ultim criteri, i havent esmenat els dos anteriors, fem servir gairebé el mateix procés. Sempre hem fet servir una eina per saber si s'ha de fer intercanvi entre els dos valors o no. En el primer cas el canvi es realitza si el winrate es mes gran o mes petit, en el segon el canvi es realitza si el compareTo retorna segons quin nombre, en aquest tercer cas la cosa es complica. Com que per el tercer cas hem de tenir en compte la nacionalitat, el winrate i per ultim el kda, mantenim aquest ordre de prioritats a la partitionKDA. Primerament comparem la nacionalitat. En cas que aquesta no estigui correctament endreçada les endreça. Quan la nacionalitat és igual entre dos jugadors, es crida la funció "escalaU", que transforma el valor que se li pasa a un radi de [0,1], per així poder fer una comparació entre pesos (balanç entre winrate/kda). Un cop s'han determinat els pesos, es compara quin dels dos té el pes més gran i s'ordena segons aqeust criteri (fent un

swap).
Un cop reendreçat el array, igual que en els anterior apartats, es retorna la posició del inici del array per acabar de endreçar els arrays restants ja que el quickSort es una crida recursiva i fa us del valor de la partició per dividir els nous arrays.

2.2.2 Merge Sort

Al igual que hem fet amb el Quick Sort, el Merge Sort el dividirem segons quin tipus d'input vulguem ordenar, és a dir, Equips o Jugadors.

- Merge Sort Equip (Per WinRate):

Aquest mètode d'ordenació necessita inicialment d'un array d'equips, el seu principi(índex l de left) i el seu fi(índex r de right). Un cop es tenen aquests valors, es mira si Left és més petit que Right (per tant, que l'array encara es pot dividir en més parts). De ser així, es calcula on estaria el punt mig i de manera recursiva es van partint les dues mitats resultants. Un cop dividit en totes les parts possibles, es crida la funció "sortEquip". Aquesta funció s'encarrega d'ordenar segons el WinRate la porció de l'array que se li ha passat.

- Merge Sort Jugador (Per Nacionalitat):

La filosofia és la mateixa, amb la única diferència que ara tenim llistes de jugadors, i en contes de mirar índex, mirem la grandària de la llista inicial. Ara en contes de tenir índex, tenim fues llistes auxiliars, que seran directament les llistes que hem anat dividint, i anirem ordenant segons la seva Nacionalitat amb la funció compareTo(), que ens retorna un enter, indicant si és més petit, igual o més gran. En el cas que les nacionalitats siguin diferents, s'ordenarà per Nacionalitat. Si les Nacionalitats són iguals, llavors mirarem el nom dels Jugadors, i farem el mateix procés de cridar la funció compareTo(), per acabar sabent quin és alfabèticament el que ha d'anar primer.

- Merge Sort Jugador (Per Combinació de Prioritats):

Com sabem, el merge sort es una funció recursiva. Sempre que es crida a si mateixa es passa la divisió en 2 de les llistes creades aconseguint dividir totes les llistes entre 2 fins a no poder dividir més. Un cop estan totes divididas en aquest criteri, a la funció sortJugador, la encarregada de "pujar" en aquest arbre de divisions i anar-les endreçant, es recorren les llistes i, igual que en alguns sorts anteriors, fem servir la eina compareTo per saber si s'haurien de intercanviar posicions els dos items de la llista. En cas que el compareTo ens doni 0, indicant que la nacionalitat es la mateixa, es crida la funció "escalaU", que transforma el valor que se li pasa a un radi de [0,1], per així poder fer una comparació entre pesos (balanç entre winrate/kda). Un cop s'han determinat els pesos, es compara quin dels dos té el pes més gran i s'ordena segons aqeust criteri (fent un swap).
Com que estem modificant les llistes que es passen a la funció sortJugador, aquestes es modifiquen per "referència" per tant no cal retornar re i ja s'està canviant el seu valor.
Considerem important mencionar també com després de recórrer totes les dues llistes tenim 2 whiles, un per aList i un per bList que son els encarregats de comprovar els últims valors que puguin quedar residuals a les llistes i desar-los a la llista que "retorna" el sort jugador.

2.2.3 Bucket Sort

Igual que per altres mètodes de ordenació per implementar el bucket hem hagut de declarar dues funcions per separat. El bucketsortEquips i el bucketSortJugador. Això es degut a que el primer, encarregat de endreçar els equips pel seu winrate, només endreça objectes de tipus Equip mentre que els dos segons criteris demanen endreçar objectes de tipus Jugador, i per evitar problemes en la programació i no barrejar tipus de arrays hem creat dues funcions per separat.
A grans trets, el bucket funciona creant un array que anomenem "bucket", valgui la redundància, que es tan gran com el màxim valor que prengui els nombres que volem endreçar. Posem per exemple que el nombre màxim que podem endreçar te un valor de 87. Doncs el nostre bucket seria de tamany 87+1, 88. Això permet que poguem desar cada valor a la casella corresponent al seu valor i després simplement recórrer aquest array en ordre i es mostrarien els nombres en ordre. Això es suposant que no hi ha valors repetits i sense tenir en compte els valors decimals. Però això està esmenat amb més detall als apartats pertinents.

- Bucket Sort Equip (Per WinRate):

Pel winrate, com sabem, tenim un nombre de tipus float. És a dir, un nombre decimal que pot o no tenir part enter i després molt probablement te decimal. És per això que pel nostre bucket hem creat una matriu de Equips. Aquesta matriu, partint del funcionament bàsic del bucket que hem esmenat anteriorment funciona de la següent manera: La primera dimensió de la matriu correspon a aquest array en el que desarem cada nombre pel seu valor enter. Pel que la primera dimensió te de tamany el valor màxim enter que prenen els winrate. Per trobar aquest valor hem fet ús de una funció pròpia que hem anomenat "trobaMaxValWR". La segona dimensió d'aquesta matriu seria la que contindria tots els nombres que tinguin igual la part entera per tant a la casella [83] tindríem tots els 82, algun nombre. Entesa la utilitat de la nostre matriu de jugadors el que segueix es bastant fàcil. La primera dimensió la emplenem fent un cast a int dels valors dels winrate i desant allà els diferents jugadors. Un cop emplenades, hem de endreçar les columnes correctament pels seus decimals i es per això que les recorrem una a una i anem fent un swap en cas que es necessiti. Així és com aconseguim la matriu endreçada. Un cop tenim la matriu endreçada es qüestió de recorre-la en ordre i anar desant les dades al array inicial.

La unica cosa que hem hagut de tenir en compte es, com que hem hagut de declarar la matriu desde un inici i és estàtica, i ha columnes en les que potser no s'han omplert tots els valors possibles i que, per tant, hi ha valors que s'han quedat a null. Això no és problema si ho tenim en compte i es que, tant en el moment de endreçar les columnes pels decimals com alhora de recórrer la matriu i desar-ho al array original afegim una condició de que el valor actual no sigui null.

- Bucket Sort Jugador (Per Nacionalitat):

Entès el winrate la nacionalitat funciona de manera molt similar. La unica diferència es que per aquest apartat ho hem dividit en més funcions per poder després aprofitat la funció que hem anomenat "ordenaNacionalitat" més endavant. Per endreçar les nacionalitats vam haver de inventar una manera de considerar les nacionalitats nombres perquè, igual que pel winrate accediem a la casella[winrate] i aquesta contindria tots els valors coincidents amb el seu valor de casella, volíem aconseguir que a cada casella hi hagués tots els jugadors de la mateixa nacionalitat. No només això però

també volíem que aquestes estiguessin endreçades. Per fer-ho el que vam fer va ser crear un array de strings que hem anomenat "nacional". Aquest array te un tamany del nombre total de nacionalitats i, a la funció omple-Nacional, l'omplim de les diferentes nacionalitats que hi ha. Després, a la funció endreçaNacionalitats, endrecem les nacionalitats del array de strings de manera molt bàsica, recorrem el array i anem fent compareTo en cas que s'hagin de intercanviar es realitza un swap.

Finalment després de haver creat i endreçat aquest array de Strings tenim un array on està el nom de cada nacionalitat, en ordre. Amb aquest array creat el que fem és començar el nostre bucket, la nostre matriu de jugadors. Però aquesta vegada la casella actual de la primera dimensió de la matriu a la que desar el jugador actual la desem de forma diferent, cridem a la funció casellaNacionality. Aquesta rep el array de strings i el jugador actual i retorna a quina casella es troba la nacionalitat del jugador actual i un cop tenim la casella, desa el jugador actual en aquella casella. La segona dimensió de la matriu es de tamany màxim de jugadors en una sola nacionalitat. Pel que un cop fet això tenim a la casella [0] tots els jugadors de la primera nacionalitat, a la casella [1] els de la 2a, i així fins recórrer totes.

Un cop creada la matriu de jugadors endreçats per nacionalitats aquesta es retorna a la funció principal "bucketSortJugador" per acabar de endreçar els de la mateixa nacionalitat pel seu nom. Això ho fem simplement recorrent la primera dimensió de la matriu i endreçant cada columna com a un array per separat. Per endreçar-la fem us de la sentència compareTo un altre vegada per comparar els noms i en cas que requereixin un canvi es fa el canvi pertinent amb un swap.
Finalment, igual que pel winrate, es recorre aquesta matriu de jugadors tornant a desar-los al array original i, ja que ja estan en ordre a la matriu desant-los en ordre al array els està endreçant.

- Bucket Sort Jugador (Per Combinació de Prioritats):

Aquest últim criteri ha estat probablement el més difícil de fer pel bucket ja que hem hagut de fer un conjunt de tot.
El que fem primerament es fer us de la funció anteriorment esmenada per que ens retorni una matriu de jugadors ja classificats per nacionalitat. I igual que a la nacionalitat recorrem cada una de les columnes endreçant-los pel nom, aquí recorrem cada una de les columnes però es crida la funció "escalaU", que transforma el valor que se li pasa a un radi de [0,1], per així

poder fer una comparació entre pesos (balanç entre winrate/kda). Un cop s'han determinat els pesos, es compara quin dels dos té el pes més gran i s'ordena segons aquest criteri (fent un swap).

2.2.4 Radix Sort

- Radix Sort Equip (Per WinRate):

En aquets mètode d'ordenació se'ns planteja l'inconvenient de que els números a ordenar són del tipus float, cosa que dificulta la seva ordenació. Per solucionar aquest problema el que s'ha fet és agafar el nombre amb els màxims decimals i multiplicar tots els nombres del array per 10 elevat al nombre de decimals màxim. Un cop fet això, s'han tractat el nombre com a enters. Amb els nombres enters, hem agafat el dígit de menys pes i l'hem col·locat a la posició del array corresponent, és a dir si tenim el dígit 5, estarà col·locat en la posició de l'array 5. Un cop fet això per totes les dígits, s'ha dividit el nombre entre 10 elevat al nombre de decimals màxim, per així tornar a tenir nombres decimals ja ordenats.

- Radix Sort Jugador (Per Combinació de Prioritats):

En aquesta funció seguim la filosofia del Radix Sort Equip. Primer de tot canviem tots els winrates, multiplicant-los tots per 10 elevat al nombre de decimals màxim. Abans de fer això, s'ordena per Nacionalitat, fent servir el mètode Bucket, és a dir, una matriu on la primera columna estan tots els jugadors que tinguin com a nacionalitat la primera nacionalitat, ja ordenada de la A a la Z (en el cas de la pràctica, Canada). Un cop tenim els WinRates modificats i els jugadors classificats per nacionalitat, es mira els pesos calculats a través de la funció "escalaU", que transforma el valor que se li pasa a un radi de [0,1], per així poder fer una comparació entre pesos (balanç entre winrate/kda). Un cop s'han determinat els pesos, es compara quin dels dos té el pes més gran i s'ordena segons aqeust criteri (fent un swap).

3 Comparativa d'algorismes

3.1 Cost Temporal

Per la comparativa de algorismes hem emprat la eina que hem esmenat anteriorment del temps del sistema. Hem mesurat el temps que triga cada algorisme en executar-se. A cada crida de funció tenim el startTime1, que mesura el temps en nano-segons i el startTime2 que mesura el temps en milisegons. A la imatge que tenim a continuació es pot veure com està en el codi.

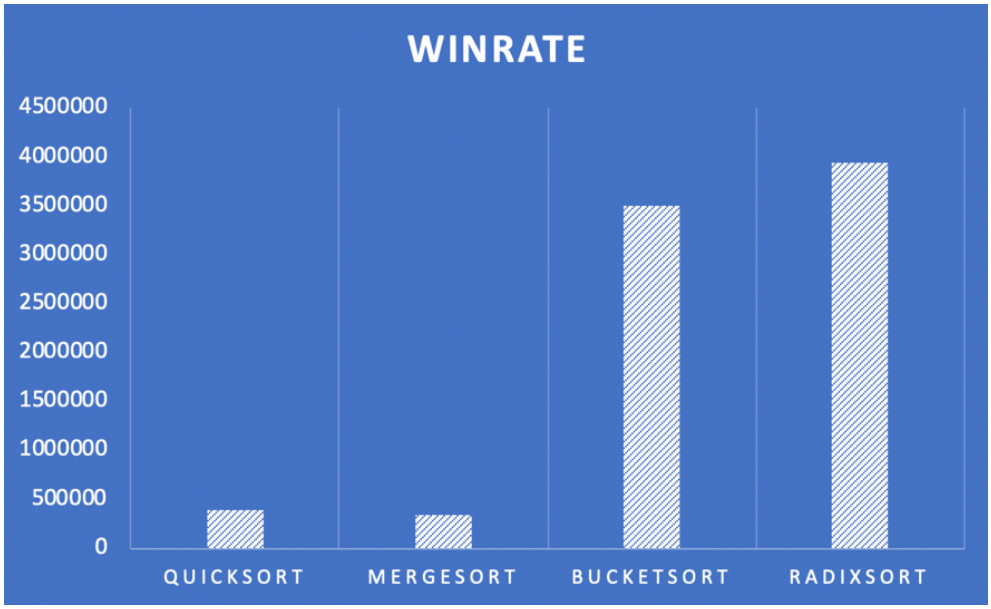
```
startTime1 = System.nanoTime();
startTime2 = System.currentTimeMillis();
switch (s) {
    case "1":
        equip.mergeSortEquip(equips, l: 0, r: equips.length - 1);
        break;
    case "2":
        equip.mergeSortJugador(j, s);
        break;
    case "3":
        equip.mergeSortJugador(j, s);
        break;
}
endTime1 = System.nanoTime();
endTime2 = System.currentTimeMillis();
```

Si que es cert que en aquest temps també està inclòs el temps que triga en fer-se el switch. Per eliminar aquest factor i altres factors que poden ser condicionants de la "eficàcia" dels algorismes hem realitzat la mitja de un seguit de temps diferents per poder tenir un valor mitg que molt probablement representi el temps que triguen els algorismes de mitja. Per tal de assolir la major precisió hem emprat el temps en ns ja que tot i que hi hagi diferència entre una mesura i un altre és molt més exacte que la mesura en ms.

Pel primer conjunt de valors hem recollit els temps de execució de cada un dels algorismes per calcular el winrate. S'han realitzat tres mostres ja que hem considerat que era un nombre de mostres suficient i que ja mostraria gairebé en la seva totalitat la realitat del temps de execució.

WINRATE				
	TEMPS 1 (ns)	TEMPS 2 (ns)	TEMPS 3 (ns)	TEMPS FINAL (ns)
QuickSort	838537	206530	124427	389831
MergeSort	235063	226848	563672	341861
BucketSort	4239783	3658219	2577888	3491963
RadixSort	4866208	3426778	3514698	3935895

A nivell numèric ja podem veure de primeres quin es el algorisme més lent en executar-se i quin és el més ràpid. El mergeSort, es el més ràpid en executar-se de mitja seguit molt properament pel quickSort. Volem recordar que estan mesurats en nano-segons pel que la diferència entre el quickSort i el mergeSort es gairebé nul·la, almenys per aquesta mostra de dades però es probable que la diferència augmenti per mostres més grans.



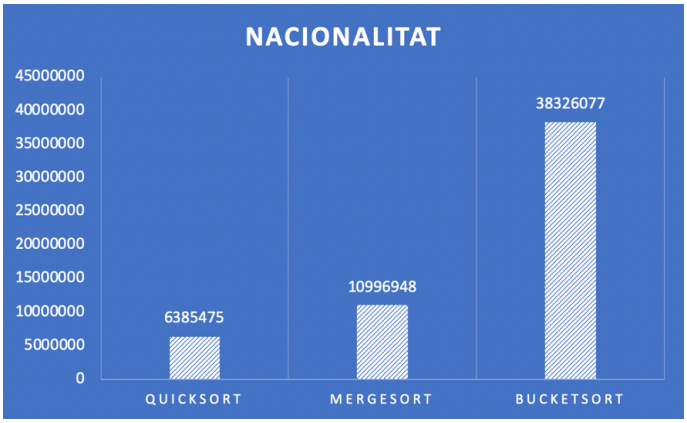
En cas que amb nombres no veguem la gran diferència ja que si que és cert que son nombres molt grans podem veure la diferència entre cada un en aquesta gràfica. El mergeSort i el quickSort son clarament els més ràpids en executar-se i el bucket i el radix els més lents sent el radix el més lent de tots.

Anem a veure la diferència en quant a la nacionalitat. Per la nacionalitat la taula té una forma similar. Com que no es demanava implementar el radix per la nacionalitat aquest no ha estat inclòs. Novament, veiem com el quickSort és el més ràpid i el bucket el més lent.

NACIONALITAT

	TEMPS 1 (ns)	TEMPS 2 (ns)	TEMPS 3 (ns)	TEMPS FINAL (ns)
QuickSort	5580989	7646097	5929340	6385475
MergeSort	13609303	9066592	10314948	10996948
BucketSort	30322603	39331883	45323746	38326077

A la gràfica veiem com aquí la diferència entre el merge i el quick comença a ser més significativa tenint en compte que aquests valors ja son valors mitjos pel que son una representació bastant acertada dels seus temps de execució. El que és indiscutible és que el bucket triga molt més en executar-se.

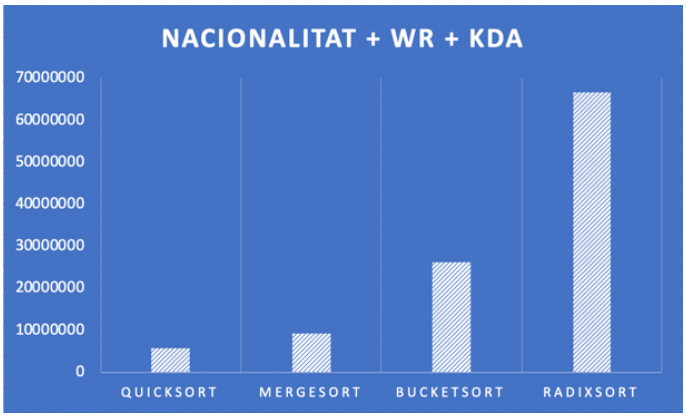


Finalment, el tercer criteri i el que han trigat més en resoldre cada un dels mètodes de ordenació, la combinació de prioritats. Per aquest podem veure com el temps de execució augmenta gairebé exponencialment.

COMBINACIÓ (NACIONALITAT + WR + KDA)

	TEMPS 1 (ns)	TEMPS 2 (ns)	TEMPS 3 (ns)	TEMPS FINAL (ns)
QuickSort	5470853	5588969	6033813	5697878
MergeSort	10082981	8176569	9550998	9270183
BucketSort	26175065	24459293	28244279	26292879
RadixSort	65191208	68556429	66100699	66616112

Potser no es veu tan clar a la taula de valors però es veu bastant clar a la gràfica on podem veure com el merge es gairebé el doble de alt que el quick, el bucket gairebé el doble que el merge i el radix gairebé el doble que el bucket. Aquest comportament es molt similar al comportament de un cost exponencial i tot i que entre ells no tinguin cap relació si que podem veure com per una tasca com aquesta surt molt més rentable emprar el quickSort a emprar un radix. Aquesta conclusió no la hauríem pogut realitzar de no comparar els temps de execució tal i com se'ns ha plantejat al enunciat de la pràctica.



3.2 Cost Asimptòtic

En l'apartat anterior hem vist com ha evolucionat el temps d'execució depenent del mètode d'ordenació selecionat i el dataset compilat. Ara bé, com a concepte, els diferents mètodes d'ordenació tenen un cost asimpotòtic associat, exposat a continuació.

- QuickSort:

Tal i com hem vist, en l'algorisme QuickSort es requereix d'un pivot, el qual s'ordenaran els més petits a l'esquerra d'aquest i el més gran a la dreta, depenent de quin element s'agafi com a pivot(fent un swap). Un cop es fa el swap, es separa aquest en dos (com si fos un Merge Sort) i es repeteix el procés fins que està tot ordenat. També sabem que el cost asimptòtic d'anar dividint un array tindrà un logaritme relacionat.Aquest mètode d'ordenació ens fa tenir tres escenaris diferents:

1) Pitjor cas:
Aquest cas ens el trobarem quan, respecte el pivot, s'hagi de fer un swap complet. En aquest cas el cost serà de $O(n^2)$.

2) Millor cas:
En canvi, en el cas que la partició es faci per la meitat, és a dir, el pivot pugui ordenar amb normalitat i quedi al mig del array (per tant es parteix en dos arrays d'igual tamany), tindrem un cost de $O(n \cdot \log n)$.

- Merge Sort:

Aquest algorisme és molt pròxim al anterior exposat, on separa sempre l'input en dos fins que ens quedem amb casos trivials. Aquest fet fa que ens aparegui un cost constant, ja que per molt llarg que sigui l'array a ordenar, aquest sempre s'haurà d'anar separant en 2. Així doncs tenim un cost constant de $O(n \cdot \log n)$.

- Bucket Sort:

En aquest algorisme, ja que anem assignant valors a caselles d'un array auxiliar (representant el dígit que agafem), aquest cost ens dependrà de com es reparteixin els números en les caselles:

1) Pitjor cas:
Com a pitjor cas, tindrem la situació en que el Bucket Sort consideri que tots els elements vagin a la mateixa "bucket", cosa que ens farà apareixer un cost de $O(n^2)$.

2) Millor cas:
En el millor cas, ens trobem en la situació en que els dígits s'han pogut distribuir de manera equitativa a tots els "buckets", per tant es pot fer una ordenació de cost O(n+k), on k representa el número de caselles de l'array auxiliar, on col·loquem els numeros segons el seu dígit.

- Radix Sort:

En el Radix Sort, el cost dependrà del número de dígits màxim amb el que contem, sent aquest k. També cal tenir en compte que interiorment implementa un Count Sort, per ordenar cada dígit. Un cop es té en compte aquestes variables, podem dir que el comportament asimptòtic del radix Sort serà de O(kn).

3.3 Afinitats

Ara, pasarem a comparar els diferents mètodes d'rodenació, ja que com hem pogut veure, entre ells tenen una gran afinitat. Aquesta afinitat es veu reflexada en els costos i en situacions on un mètode es comporta igual que un altre.

1) MergeSort - QuickSort

Aquests dos mètodes tenen una cosa en comú, ja que els dos divideixen entre dos els arrays que reben com input. Aquest fet fa que els seus costos siguin molt pròxims, amb la ínica diferència que el Quick Sort pot tardar més asimptòticament parlant. Aquest fet es demostra en la següent gràfica, on per arrays petits, el Quick Sort és més ràpid, pero el Merge Sort és més eficient per array més grans, ja que les probabilitats de que es produeixi el pitjor cas en el Quick Sort augmenten.

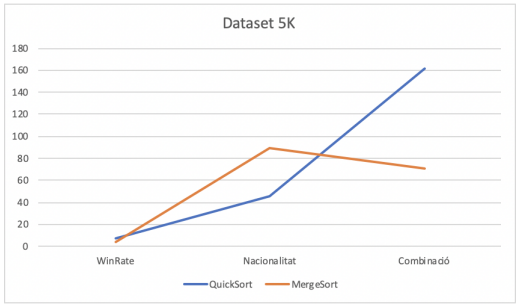


Figure 1: Gràfic escalat en Segons

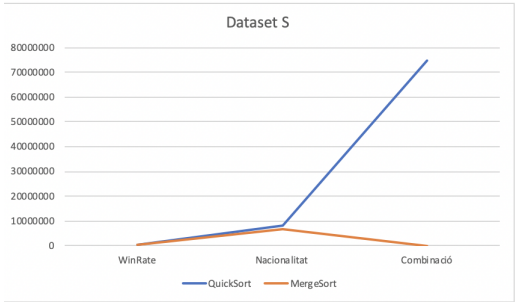


Figure 2: Gràfic escalat en Microsegons

2) BucketSort - RadixSort

Aquests dos algorismes tenen semblança, ja que els dos miren els dígits de cada número del input que se li pasa. En aquest cas, podriem arribar a implementar un Radix amb un Bucket, si cada cop que separem per dígits fessim un Count Sort. A línies generals, el bucket sort és més ràpid, per+o uutilitza més memòria, cosa que en segons quins projectes caldria tenir en compte. També cal mencionar que el Bucket Sort funciona millor per arrays més grans.

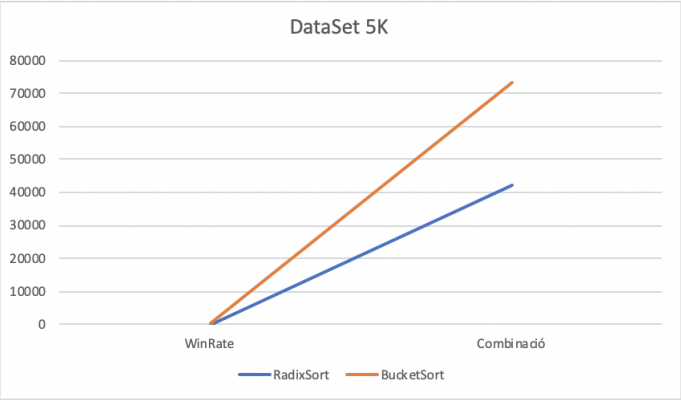


Figure 3: Gràfic escalat en Microsegons

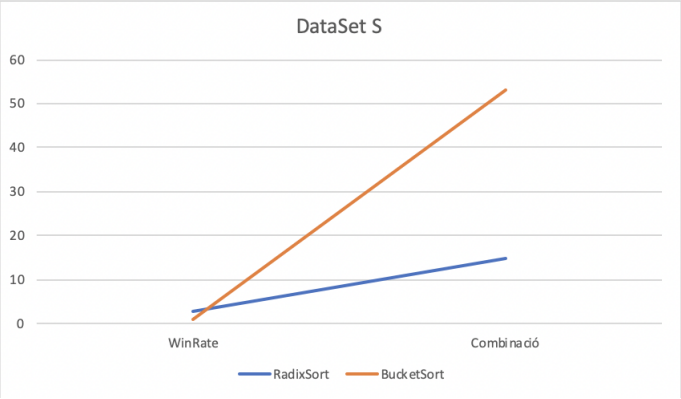


Figure 4: Gràfic escalat en Microsegons

4 Mètode de proves utilitzat

En quant al mètode de proves es resumeix en un seguit de comprovacions. Per comprovar que el que estem fent estava bé hem emprat dues eines que ens ha facilitat el IDE que hem triat. Una ha estat la eina de debug del IDE que ens ha permès veure quins valors prenien les nostres variables a diferents instants de temps per poder veure en quin moment podia fallar el nostre codi i saber com solucionar-ho.

El segon mètode ha estat el us de la compilació i execució del codi i per poder veure els errors simplement hem anat printant per pantalla algunes variables que considerem claus i això ens ha permes seguir el flow del programa i veure on fallava. Finalment en quant a la comprovació de les variable hem contrastat els resultats dels diferents mètodes de ordenació entre si i nosaltres mateixos hem obert els .json per cercar alguns valors clau per acabar de comprovar que els resultats llegits pel nostre codi eren correctes.

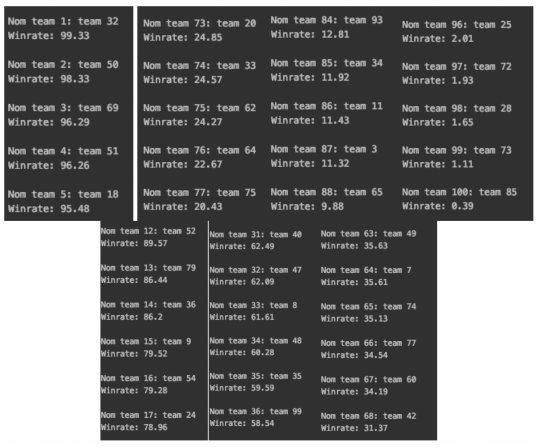
5 Resultats

Per mostrar el correcte funcionament del programa hem fet el següent: Hem triat un seguit de dataseets significatius, de cada dataseet mostrem les dades endreçades per winrate, nacionalitat i la combinació de nacionalitat, winrate i KDA. I per cada un d'aquests criteris de ordenació tenim els diferents mètodes emprats. De cada mètode mostrem el seu temps de execució calculat amb el System.nanoTime() i el System.currentTimeMillis() dues funcions que ens permeten calcular el temps de execució en ns i en ms. El temps en s el calculem a partir d'aquests valors anteriors.

I finalment, per demostrar que s'ha ordenat afegim també algunes imatges de alguns punts del dataseet endreçat per que es pugui veure com estan endreçats correctament. Com que el resultat de endreçar el winrate o la nacionalitat o la combinació ha de obtenir el mateix resultat encara que es faci amb diferents mètodes, per cada apartat adjuntem un seguit de captures i aquestes son certes per tots els mètodes de ordenació.

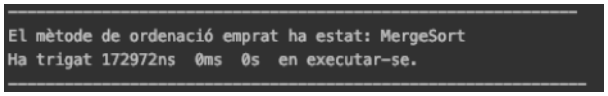
5.1 WinRate

Com podem veure per cada mètode de ordenació el exemple del dataseet veiem el nom del equip que hem numerat per tan el nom del team 1 es refereix al nom del equip amb més winrate, en aquest cas el team 32. I només mostrem el nom del equip i el seu winrate perquè considerem que aquestes son les dues dades significatives a mostrar.

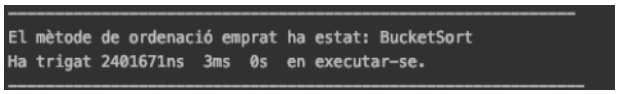


- Quick Sort:

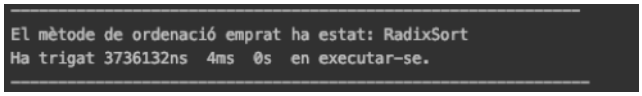
- Merge Sort:



- Bucket Sort



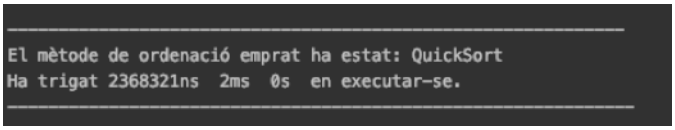
- Radix Sort



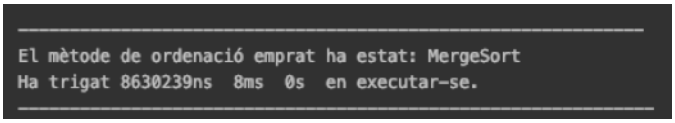
5.2 Nacionalitat

Per la nacionalitat hem decidir mostrar el nom del jugador, la seva nacionalitat i el equip al que pertany.

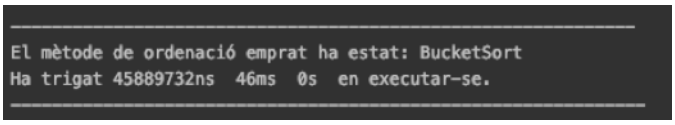
- Quick Sort:



- Merge Sort:

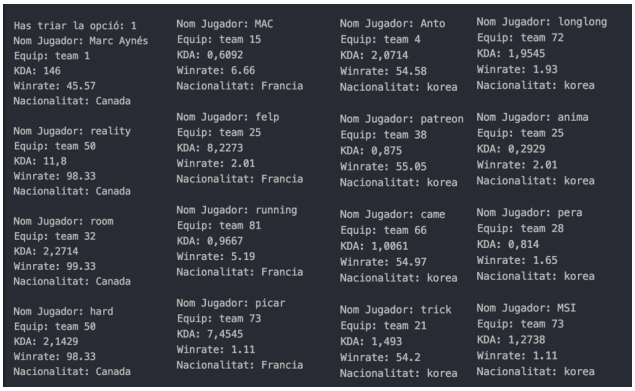


- Bucket Sort

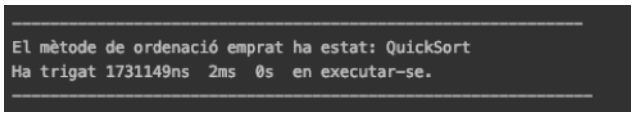


5.3 Combinació de Prioritats

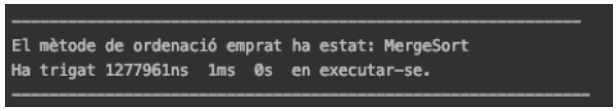
Per la mostra de la informació mostrem el nom del jugador actual, el equip al que pertany, el seu KDA, el seu winrate i, finalment, la seva nacionalitat.



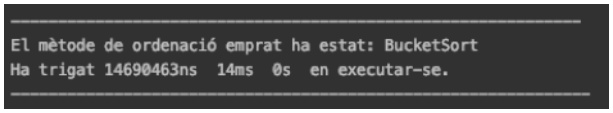
- Quick Sort:



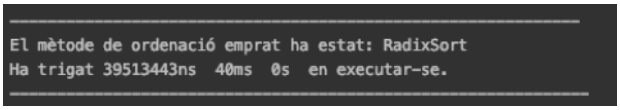
- Merge Sort:



- Bucket Sort



- Radix Sort



6 Problemes Observats

Al llarg de la pràctica el seguit de reptes que ens hem anat trobant a mesura que hem anat implementant els diferents sorts han estat molt interessants però sobretot ens han ajudat a entendre millor com plantejar els sorts i a pensar diferents maneres de afrontar "problemes" a resoldre. El criteri de ordenació que més problemes ens ha suposat ha estat el 3r, en el que havíem de endreçar primer tots els jugadors per nacionalitat, tot seguit endreçar els de la mateixa nacionalitat pel seu winrate i finalment els que tinguessin el mateix winrate els havíem de endreçar per el seu kda. En començar a plantejar aquest criteri ens vam adonar que pel criteri 1 haviem de endreçar equips pro que pel 2 i pel 3 els que se'ns estava demanant era que endrecéssim jugadors. El problema era que cada equip tenia un llistat de jugadors propi i en crear una llista genèrica perdíem el seu equip i el winrate del seu equip. Es per això que vam assignar el valor del winrate del seu equip a cada jugar i tmb vam desar el nom del equip al que pertanyia. Amb aquestes dades assignades només faltava crear una llista de jugadors i endreçar-los pel criteri convenient.

Un altre dels problemes afrontats ha estat el haver de endreçar "Floats". Donat que el bucket i el radix els dos funcionen millor si tenen integers amb els que treballar hem hagut de fer alguns canvis. Pel bucket vam haver de crear una matriu que la primera dimensió tenia la mida del màxim valor del winrate de tal manera que a la casella 5 teníem tots els 5, i a la casella 85 teníem tots els 85, . Un cop endreçada la primer dimensió falta endreçar els decimals. Es per això que aquesta segona de la dimensió de la matriu era important perquè es aquí on després endreçàvem els decimals.

Pel radix vam tenir un problema similar. Donat que el radix funciona recorrent valor per valor de un numero teniem un problema al implementar-lo per decimals ja que només ens endreçava la part entera. Vam fer molta cerca per tenir idees i alternatives però no vam acabar trobant res. Finalment vam fer una petita modificació als valors per tal de poder endreçar-los. Calculàvem quina mida tenia el numero més gran i quin nombre de decimals tenia el nombre amb més decimals. Multiplicaríem tots els nombres per aquest per que no quedés ni un sol amb decimals. Ni winrate ni kda. Els endrecem com a integers que això el radixSort ho fa sense problema i després alhora de tornar a desar-los a la llista original els tornem al seu valor original.

Un problema que ha estat comú al llarg del desenvolupament de la pràctica i que a vegades els resultats de un endreçament estaven invertits, de petit a gran i no de gran a petit. Això no es problema quan es el array de equips endreçats per winrate perquè simplement caldrà invertir-lo però quan es el 3r criteri i estan correctament endreçats per nacionalitat però dintre de la nacionalitat els winrate estan al revés i els kda bé, llavors tenim un problema. I això ens ha passat en diferents ocasions en diferents sorts. Per solucionar-ho hem hagut de implementar una funció "invertirArray" que te com a finalitat simplement invertir el ordre de les caselles del array. Sabem que hi ha una funció similar a java però que, pel que hem trobat, només funciona per llistes i, tot i que la hem emprada, no sempre necessitàvem invertir una llista. Evidentment abans de la creació de aquesta funció s'havia repassat el codi e intentat modificar segons quines sentències per assegurar que s'endrecés correctament ja que la creació de una funció així suposa un cost major alhora de executar el algorisme pro davant de la impossibilitat de solucionar-ho ens vam veure obligats a pensar en alternatives.

Hi ha una eina que es veurà repetida al llarg del codi i es la nostre variable "criterio". Aquesta variable és o un string o un integer i la hem emprada per, com diu el nom, distingir entre criteris. Hi ha vegades que emprem la mateixa funció per endreçar en funció de 2 coses diferents i per fer-ho saber al codi hem creat aquesta variable auxiliar que li indica quin criteri aplicar a la ordenació. Això ho considerem a la part de problemes ja que inicialment haguéssim hagut de crear o diferents funcions pels diferents casos o simplement repetir molt el codi però vam considerar que una implementació així no era tan òptima com la que podíem assolir amb el "criteri".

Per últim una situació en la que ens hem trobat però que ha resultat tenir una solució fàcil és: Com que el nostre programa es troba en un bucle permetent que trii diferentes mètodes de ordenació i criteris en una mateixa execució ens vam trobar que, evidentment, aplicar el mateix mètode de ordenació dues vegades seguides per algun motiu era més òptim la segona vegada i es que la llista ja estava endreçada de la primera. Per mantenir el funcionament del programa fidel al que hauria de ser i que s'endreçi cada vegada, a cada volta del bucle tornem a crear el array de equips i/o llista de jugadors per tal de que el algorisme les torni a endreçar novament.

7 Conclusions

7.1 Personals

En aquest treball hem après com els quatre diferents mètodes d'ordenació funcionen i quin es la seva manera d'ordenar els inputs. També hem observat com el simple fet d'ordenar un array o una llista es pot portar a un punt de complexitat prou elevat, i que d'això se'n derivi una codificació extensa. A aquest fet se li ha de sumar que els arxius lliurats (JSON) consten d'una extensió considerable, cosa que s'aproxima al volum de dades reals que ens podríem trobar en el món laboral.

Gràcies a que ens han donat llibertat en el llenguatge, hem pogut ampliar els nostres coneixements envers el llenguatge Java orientat a objectes, llenguatge el qual encara no estàvem familiaritzats del tot però que ens resultava més còmode d'operar. També cal mencionar que hem codificat els mètodes d'ordenació de manera recursiva, cosa que ens ha permès veure una aplicació real i potent de la recursivitat.

Per últim, també cal mencionar que gràcies a aquesta pràctica ens han donat a conèixer l'eina Overleaf, per fer memòries amb LaTex. Llenguatge el qual ens ha costat adaptar-nos però ha resultat molt útil en el transcurs de la redacció de la pràctica.

7.2 Tècniques

Pel que fa l'apartat tècnic, s'ha observat com els temps d'execució de cada mètode d'ordenació, ha anat evolucionant de manera molt diferent, cosa que ens porta a veure les diferències que tenen pel que fa al rendiment e optimització d'aquests.

Si mirem el temps d'execució en el criteri del WinRate, podem observar com el Quick Sort ho fa de manera molt més ràpida que no pas els altres mètodes d'ordenació. Això es deu a que el Quick Sort, per a fitxers relativament petits, pot fer el seu mecanisme (agafar un element com a pivot i ordenar segons el valor d'aquest) molt més ràpid que no pas els altres mètodes d'ordenació. També cal mencionar que el Quick és totalment recursiu, cosa que agilitza molt el temps d'execució, ja que en els dos últims mètodes implementats s'han codificat amb un ús de bucles més gran que no pas el Quick.

Pel que fa la Nacionalitat, podem observar com el Quick i el Merge fan la tasca d'ordenar relativament més ràpid que el Bucket. Això és degut a que el Bucket ha d'anar classificant cada Nacionalitat i Nom dels jugadors segons les lletres del abecedari, quedant dos arrays de 27 posicions cada array, cosa que fa la seva lectura un procés més lent que no pas aplicant l filosofia del Quick i el Merge.

Per últim, la combinació de prioritats podem observar com el Merge és el més eficient, ja que el Quick tarda més en fer tres vegades la cerca i ordenació respecte l'element pivot (una pel Winrate, un altre pel Kda, i finalment per la Nacionalitat), que no pas el Merge Sort, ja que les separacions dels arrays en arrays més petits l'assoleix més ràpid per tal i com està codificat.

En conclusió, s'ha pogut veure com el Quick Sort i el Merge Sort han estat els més eficients (en el nostre cas) degut a la seva senzillesa de codi respecte al Bucket Sort i el Radix Sort. Aquests dos últims han tardat més degut a la seva codificació, ja que s'han requerit de més funcions i més bucles, cosa que augmenta els seus temps d'execució. Per tant, a nivell conceptual els quatre mètodes funcionen, però els seus costos temporals es veuen afectats degut a la complexitat del codi que els implementa.

8 Bibliografia

References

[1] QuickSort – GeeksforGeeks
https://www.geeksforgeeks.org/quick-sort/

[2] QuickSort GeeksforGeeks – YouTube
https://www.youtube.com/watch?v=PgBzjlCcFvc

[3] MergeSort – Baeldung
https://www.baeldung.com/java-merge-sort

[4] MergeSort – GeeksforGeeks
https://www.geeksforgeeks.org/merge-sort/

[5] Sorting Algorithms in Java – StackAbuse
https://stackabuse.com/sorting-algorithms-in-java/

[6] BucketSort – GeeksforGeeks
https://www.geeksforgeeks.org/bucket-sort-2/

[7] BucketSort – Baeldung
https://www.baeldung.com/java-bucket-sort

[8] BucketSort – Programiz
https://www.programiz.com/dsa/bucket-sort

[9] BucketSort – GrowingWthTheWeb
https://www.growingwiththeweb.com/2015/06/bucket-sort.html

[10] BucketSort – JavaRevisited
https://javarevisited.blogspot.com/2017/01/bucket-sort-in-java-with-example.html

[11] RadixSort – CodeReview
https://codereview.stackexchange.com/questions/129030/radix-sort-in-java

[12] radixSort – java67
https://www.java67.com/2018/03/how-to-implement-radix-sort-in-java.html

[13] RadixSorti – c- sharpcorner
https://www.c-sharpcorner.com/UploadFile/fd0172/radix-sort-in-java/

[14] RadixSort – ICS UCI
https://www.ics.uci.edu/ eppstein/161/960123.html

[15] RadixSort – Baeldung
https://www.baeldung.com/java-radix-sort